# Circuit Intermediate Representation

Last Updated: 2022-12-01

## Contents

# 1 Introduction

In Phase II of the SIEVE Program, we are currently defining two Intermediate Representations called Circuit IR and Translation IR and a Circuit Configuration Communication file. This document will only present the current version of Circuit IR.

The IR is an interaction between the frontend and the backend portions of a Zero Knowledge (ZK) proof pipeline. The frontend transforms high level statements in a target domain into the IR. It is the producer of the three resources which are the subject of this document. The backend is the consumer of them: it is an interaction between a Prover and a Verifier, where the Prover wishes to prove a statement to the Verifier without revealing a secret component of the proof.

## 1.1 Overview

At a high level, this interaction involves three resources provided by the frontend and used by the backend, as the fact is being proven.

- Relation (Circuit or Translation IR) – a mathematical relationship between the inputs.
- Public inputs – inputs to the relation given to both the Prover and Verifier.
- Private inputs – inputs to the relation given only to the Prover.

Version 2 of the SIEVE IR seeks to resolve the conflict between the necessity for frontend and backend interoperability and the variety of design choices, capabilities, and requirements in each system. The SIEVE program has seen everything from C++ libraries through R1CS as viable backend-specific IRs.

To resolve this issue, the SIEVE IR introduces two types of relation: the Circuit-IR and the Translation-IR.

- The Circuit-IR is defined by flat lists of gates and wires; functions may be defined and reused within the circuit.
- The Translation-IR is a program which outputs a relation in the Circuit-IR format. The backend is given some amount of control over how the Translation-IR is translated, and is free to reimplement common or even standardized libraries.

At QEDIT, we decided to focus on the Circuit IR for the following reasons

- it is easily plugable to frontends/backends because it is close to existing representations (e.g. R1CS, PLONK)
- it supports complex features and it is easily extensible thanks to plugins
- it is a simple representation that simplifies newcomers' adoption and maintainability

## 1.2 Multi Field Circuits

To most practitioners of ZK, a single prime field is chosen at the beginning of a proof and used throughout. However, for some applications, it is desirable to use multiple primes for different elements within a single larger proof. For example, a large and expensive prime may be needed to verify public-key signatures, while a medium sized prime is necessary for large scale business logic.

To accommodate these applications, the IR must allow for multiple fields within a single relation. To the frontend, each field must describe the type of a wire, while to the backend, these wires actually belong to multiple independent proofs. An analogy to the real world might be a circuit card with transistor logic on one side and high voltage on the other.

Occasionally information from one field will be required in another. The IR models this using a conversion gate with inputs in one field and outputs in another. To continue the analogy, a relay would allow information to flow from transistor logic into high voltage, or in reverse, an analog-digital converter. In ZK, methodologies must be developed and used to show the equivalence of inputs and outputs across independent proofs or even across different proof systems.

# 2 Headers

All SIEVE IR files start with a header. The header contains: a version number for quick recognition of the IR and a resource type (`circuit`, `translation`, `public_input`, `private_input`, or `configuration`). Each resource type may define additional header declarations that appear after the version and resource type. Here is an example header.

```
version 2.0.0;
private_input;
```

This indicates the file is a private input resource for version 2.0.0 of the SIEVE IR.

# 3 Circuit IR

## 3.1 Header

As stated in section 2, the Circuit IR header starts with the version number and resource type.

```
version 2.0.0;
circuit;
```

Next, all plugins (Section 4), types, and conversion gates that are used in the file are declared up front in the header, before the `@begin` keyword. The order of these declarations must be sorted. Plugins must appear first, followed by types, and then conversion gates come last.

A type declaration specifies the types that are used in the rest of the file. Most types specify a `field` and indicate the field's characteristic (prime). Type declarations also implicitly specify a type-index, assigned incrementally as each type is specified. The maximum number of types that can be defined is 256.

```
// index 0: Boolean
@type field 2;
// index 1: 2^61 - 1
@type field 2305843009213693951;
// index 2: 2^255 - 19
@type field 57896044618658097711785...;
```

This example declares three types for the Boolean, $2^{61} - 1$, and $2^{255} - 19$ fields, respectively indexed by 0, 1, and 2.

All conversion gates used in the file must be declared in the header. Each conversion gate declaration specifies which fields are converted between and how many of each field is input and output. Here are a few examples.

```
// Convert Booleans to Mersenne61 and back
@convert(@out: 1:1, @in: 0:61);
@convert(@out: 0:61, @in: 1:1);
// Convert Mersenne61 to 25519 and back
@convert(@out: 1:5, @in: 2:1);
@convert(@out: 2:1, @in: 1:5);
```

Conversion gates are fully specified in section 3.5.

## 3.2 Public and Private Inputs

Inputs to the circuit are provided through separate resources (See Section 5) and accessed as streams. There are two streams per type, one for public inputs and one for private (prover only) inputs.

Stream access uses the syntax `$output_wire <- @public([type_idx]);` for public inputs and `$output_wire <- @private([type_idx]);` for private inputs. An item is read from the stream corresponding to the type index and assigned to the output wire. If the type index is not specified, it defaults to zero.

## 3.3 Memory Management

Backends that consume SIEVE IR are highly optimized. To minimize their overhead in managing memory, the IR exposes primitives to allocate and deallocate ranges of memory. There are strict restrictions on memory management: many operations require a range of input or output wires that are not only consecutive but also are part of the same allocation. This allows optimized backends to ensure that the input or output wires are stored in contiguous memory. Two wires may have consecutive wire-numbers, but be non-contiguous in memory, depending on the backend's internal memory management strategy.

Each type is given its own numbering space, with wire-numbers in the range of $0...2^{64} - 1$. Most directives will use a type-index parameter to select in which type, and in which numbering-space, they will act. For example, `0: $123` and `1: $123` may both be defined, with each wire residing in different numbering space due to their different types.

To allocate a range of wires explicitly, the `@new([ type_idx: ] $first ... $last);` directive may be used. This creates a new allocation containing exactly the wire numbers `$first`

`...` `$last`, but does not assign values to those wires. Reads from uninitialized wires is a failure of resource validity. The new allocation must not overlap any previous allocation. If the type index is not specified, it defaults to zero.

```
@new(1: $100 ... $200);
```

Directives that assign to wires will implicitly allocate the output wires if needed. For a directive that assigns to a range of output wires `type_idx: $first ... $last`, if all wires in the range are unallocated, it first creates an allocation as by the directive `@new(type_idx: $first ... $last)` and then assigns to the newly-allocated wires. If, instead, any of those wires were previously allocated, then they all must be part of a single allocation; if only some of the wires in the range were allocated, or different wires are part of different allocations, this is a failure of resource validity. This applies even for single wires; a single output wire such as `type_idx: $wire` is treated as a one-element range `type_idx: $wire ... $wire`. If a directive has multiple output wire ranges, each is handled independently, even if the ranges use consecutive numbers.

```
// Assume no wires are previously allocated.

// Implicitly allocates the range $100 ... $199
$100 ... $199 <- @call(foo1);

// Implicitly allocates the single wire $200
$200 <- @call(foo2);

@new($300 ... $399);
// All wires $300 ... $399 are already allocated, so this does
// not implicitly allocate.
$300 ... $399 <- @call(foo3);

@new($400 ... $449);
// Error: only some of the wires are allocated.
$400 ... $499 <- @call(foo4);

@new($500 ... $549);
@new($550 ... $599);
// Error: wires are not all part of a single allocation.
$500 ... $599 <- @call(foo5);

// This creates a separate allocation for each output range.
$600 ... $649, $650 ... $699 <- @call(foo6);
```

The `@delete` directive deallocates wires with the form `@delete([ type_idx: ] $first ... $last );`. All wires in the range `type_idx: $first ... $last` must be assigned and must not have been previously deleted. The range may span multiple allocations, but it must cover each allocation in full; deallocating only part of an allocation is a failure of resource validity. If the type index is not specified, it defaults to zero.

```
// Set up some allocations
@new(1: $100 ... $199)
@new(1: $200 ... $299)
// Implicit allocations are treated the same as @new
$300 ... $399 <- @call(foo)

// Error: @delete includes wires that were never allocated
@delete(1: $300 ... $499)

// Error: @delete covers only part of the $300 ... $399 allocation
@delete(1: $300 ... $310)

// Delete the entire $100 ... $199 allocation
@delete(1: $100 ... $199)

// Delete the $200 ... $299 and $300 ... $399 allocations
@delete(1: $200 ... $399)
```

Once a wire has been deleted, its wire number may not be reused and it may not be deleted again.

**IMPORTANT** Notice that the form of nearly all ranges in the IR is `first ... last` rather than `first ... length`. Ranges are inclusive on both ends.

## 3.4 Standard Gates

The form of most standard gates is `$out <- gate_name([ type_idx: ] $left_in, $right_in);`. For a circuit to be well-formed, the type index must refer to a field type. The type index is optional and defaults to type `0` when omitted. Other gates have variations on this, and are described as necessary.

- `@add` arithmetic addition
- `@mul` arithmetic multiplication
- `@addc` arithmetic addition by a constant
  - Has the form `$out <- @addc([ type_idx: ] $left_in, < right_constant >);`
- `@mulc` arithmetic multiplication by a constant
  - Has the form `$out <- @mulc([ type_idx: ] $left_in, < right_constant >);`
- Copy the input wire to the output wire

  - Has the form `$out <- [ type_idx: ] $left_in;`

- Assign the input constant to the output wire
  - Has the form `$out <- [ type_idx: ] < left_constant >;`

- `@assert_zero`
  - Has the form `@assert_zero([ type_idx: ] $wire);`

For simplicity boolean gates are replaced with mathematically equivalent arithmetic operations. This table summarizes alternative gates.

| Boolean Gate | Arithmetic Replacement |
|---|---|
| `@and` | `@mul` |
| `@xor` | `@add` |
| `@not` | `@addc(x, <1>)` |

For a circuit to be well formed, two rules must be obeyed when using and assigning wires. First, **topological ordering** requires that when a wire is used as the input to a gate, it must have been previously defined by an earlier gate in the scope. Second, **single static assignment (SSA)** requires that within a scope a particular wire is never redefined after its original assignment, even if it removed with the `@delete` directive.

## 3.5 Conversion Gates

Conversion gates enable conversion of wires from one field to another. Conceptually a list of wires in field A is converted to a list of wires in field B. Within the circuit, a conversion gate has the form:

```
out_type_idx: $out_first [... $out_last] <= @convert(
    in_type_idx: $in_first [... $in_last]);
```

The conversion's fields and number of wires must match a conversion specification from the front matter. If it is not the case, there is a resource invalidity. Here is an example that uses conversion gates:

```
version: 2.0.0;
circuit;
// field 0: Boolean
@type field 2;
// field 1: 2^61 - 1
@type field 2305843009213693951;
// field 2: 2^255 - 19
@type field 57896044618658097711785...;
// Declare used convert gates
@convert(@out: 1:1, @in: 0:61);
@convert(@out: 1:5, @in: 2:1);
```

```
@begin
    ...
    // convert Booleans to a single Mersenne61
    1: $0 <- @convert(0: $1 ... $61);
    // convert a single 25519 to 5 Mersenne61s
    1: $1 ... $5 <- @convert(2: $0);
    ...
@end
```

The input range `in_type_idx: $in_first ... $in_last` must be part of a single allocation. The output range `out_type_idx: $out_first ... $out_last` must either be part of a single allocation or be unallocated; if it is unallocated, the range will be implicitly allocated, as with `@new`.

Here, we define in detail the specification of a `@convert` gate. Inputs and outputs are expressed in big endian representation. To convert p wires $x_1 ... x_p$ in field A into q wires $y_1 ... y_q$ in field B, we first convert the p wires in field A into a natural number $N = \sum_{i=1}^{p} x_i \times A^{p-i} \mod B^q$. Then we represent $N$ into q wires in field B $y_1 ... y_q$: $N = \sum_{i=1}^{q} y_i \times B^{q-i}$.

## 3.6 Function Gates

Function gates define a sub-circuit which may be reused multiple times. The function's outputs and inputs are given as ranges mapped sequentially, and by type, into the function's scopes. In the function's signature, each range is defined by a length and a type index. When the function is invoked, each range is mapped into its scope incrementally from 0.

Function declaration and invocation have the following forms:

```
@function(function_name,
    [@out: out_type_idx_0: out_field_count_0
           [, out_type_idx_n: out_field_count_n],]
    [@in: in_type_idx_0: in_field_count_0
          [, in_type_idx_n: in_field_count_n],]
    )
  /* gate list */
@end

[$out_first_0 [ ... $out_last_0 ]
  [, $out_first_n [ ... $out_last_n ] ] <- ]
  @call(function_name [, $in_first_0 [ ... $in_last_0 ]
      [, $in_first_n [ ... $in_last_n ] ] ]);
```

Note that function invocations do not specify the type index for inputs and outputs since they can be inferred from the function signature.

### 3.6.1 Function Gate Example

```
@function(dot_prod_10, @out: 1:1; @in: 1:10, 1:10)
```

```
  // omitted
@end

@new(1: $0 ... $9);
@new(1: $10 ... $22);
// assign $0 ... $19

$25 <- @call(dot_prod_10, $0 ... $9, $10 ... $19);
```

The `@call` directive must have one range of input wires for each input range declared in the `@function` declaration. Each range of input wires must be part of a single allocation. Similarly, the `@call` must have one range of output wires for each output range declared in the `@function`. Each range of output wires must either be part of a single allocation or be unallocated; if it is unallocated, the range will be implicitly allocated, as with `new`.

### 3.6.2 Function Declaration Ordering and Recursion

Functions are declared at the top level of the circuit. Function names come into scope after their declaration. This prevents recursive functions and allows typechecking while processing the file as a stream. For example, the following invocation is valid.

```
@function(a) /* ... */ @end

@function(b)
  @call(a);
@end

@call(b)
```

The next example is invalid since the function `a` has not been declared and is not yet in scope when `b` is defined.

```
@function(b)
  @call(a);
@end

@function(a) /* ... */ @end

@call(b)
```

## 3.7 Example

Here is a full example of a right-triangle using the Circuit IR.

```
version 2.0.0;
circuit;
@type field 7;
@type field 127;
```

```
@convert(1:1, 0:1);

@begin
  // mod 7 hypotenuse
  $0 <- @public(0);
  // mod 7 legs
  $1 <- @private(0);
  $2 <- @private(0);

  // mod 7 is too small to square them
  1:$0 <- @convert(0:$0);
  1:$1 <- @convert(0:$1);
  1:$2 <- @convert(0:$2);

  // square them
  $3 <- @mul(1: $0, $0);
  $4 <- @mul(1: $1, $1);
  $5 <- @mul(1: $2, $2);
  $6 <- @add(1: $4, $5);

  // invert the hypotenuse
  $7 <- @mulc(1: $3, <126>);

  // assert equal
  $8 <- @add(1: $6, $7);
  @assert_zero(1: $8);
@end
```

## 3.8 Circuit Semantics and Validity

When working with the Circuit-IR there are three levels of semantics and validity to be considered. Each level builds upon the prior level.

1. **Syntactic Validity:** The IR resource is recognizable in the language defined by the IR's grammar (see appendix A).
2. **Resource Validity:** The IR resource obeys semantic rules which are falsifiable with just the single resource.
3. **Evaluation Validity:** Three IR resources (relation, public inputs, and private inputs) obey semantic rules which are only falsifiable in tandem.

While syntactic validity is important, it is easy to check using off the shelf parsing tools. The focus of this subsection is on resource validity and evaluation validity.

Each resource is checked individually for **Circuit Well-formedness** or **Stream Well-formedness** (See Section 5 for details). Circuit Well-formedness focuses on ensuring that wires are connected correctly – a "broken" wire would make the circuit poorly-formed.

**Topological Ordering** For a wire to be the input to a gate, it must have previously been assigned as an output wire within the same scope.

**Static Single Assignment** Each wire which is allocated must be assigned exactly once within its scope.

**Allocation of Range Arguments** When passing a range of wires (as either an input or an output), all wires in the range must belong to the same allocation, and the range's cardinality must match the called function or conversion gate's specification.

**Deletion of Whole Allocations** When passing a range of wires to a `@delete` directive, all wires within the range must have previously been assigned and all allocates within the range must be whole allocations. E.g. a `@delete` directive may not split an allocation into smaller portions.

To meet **Evaluation Validity**, all three resources are evaluated together, and the following conditions must be met.

**Assertions** Each input to an `@assert_zero` directive must carry the value $0$.

**Stream Length Requirement** When the end of the circuit is reached each stream has exactly zero items remaining: it must not have run out of items before reaching the end, and there may not be any extra items.

# 4 Plugins

## 4.1 Motivation

In the previous section, we describe the Circuit IR which contains the core IR functionalities. In this intermediate representation, we would like to add some (complex) features (e.g. RAM operations). Unfortunately, each update in the basic syntax forces frontends and backends to update their IR generator and parser. We would like to avoid this burden while increasing expressibility of the language. The goal of plugins is to allow IR extensions without changing the core IR.

## 4.2 Plugin Syntax

Plugins allow a circuit to refer to specific functionalities. Those functionalities are defined in a document. Only backends that have an implementation of a plugin can evaluate statements containing that plugin.

In the circuit's syntax, plugins are similar to functions except the function body is replaced by the plugin. The declaration of a plugin function starts with the signature of a function followed by the use of a `@plugin` directive with plugin parameters that includes the plugin name, the

operation name, and its generic parameters. Invocation will remain the same as for functions. A function bound to a plugin must be declared before its invocation.

The names of plugins used must be specified in the header. This ensures that backends can easily check which plugins are used and reject the circuit if needed, prior to starting circuit evaluation.

Here is an example use of a `vector` plugin that provides a `mul` operation over ranges of wires. The type index and length are given as generic arguments to `mul`.

```
...
@plugin vector;
...
@begin
  ...
  // declare the function signature with a plugin body
  @function(vec_mul_4, @out: 0:4, @in: 0:4, 0:4)
      @plugin(vector, mul, 0, 4);
  ...
  // call the vec_mul_4 plugin
  $8 ... $11 <- @call(vec_mul_4, $0 ... $3, $4 ... $7);
  ...
@end
```

Plugin operations are generic, so `@plugin` takes a list of generic arguments after the plugin and operation names. Function definitions bound to plugin operations are fully instantiated so all generic arguments are concretized. As a result, function calls (`@call`) remain non-generic. Each instantiation of a generic operation requires a separate function declaration and `@plugin` binding. Generic arguments consist of a comma separated sequence of identifiers and numeric literals. In practice, this enables generics to specify parameters like fields, lengths, and functions. Providing functions as generic arguments enables higher order operations like maps and folds.

```
/* ... */
@plugin vector;
/* ... */
@begin
  /* ... */
  // Multiple instantiations of the same plugin (vector, mul)
  @function(vec_mul_4, @out: 0:4, @in: 0:4, 0:4)
      @plugin(vector, mul, 0, 4);
  @function(vec_mul_2, @out: 0:2, @in: 0:2, 0:2)
      @plugin(vector, mul, 0, 2);

  // Higher order map operation.
  @function(plus1_0, @out: 0:1, @in: 0:1) /* ... */ @end
  @function(vec_plus1_4, @out: 0:4, @in: 0:4)
      @plugin(higher_order, map, 0, 0, 4, plus1_0);
  /* ... */
@end
```

Each plugin operation has a signature, which is defined as part of the standard for the plugin. If the backend sees a `@plugin` for a known plugin, and the signature of the operation doesn't match the signature of the function it is being bound to, then the circuit is invalid. Further, the plugin's name must have been declared in the circuit header. Either of these errors would make the circuit **poorly formed**.

Plugin operations may consume some public and private inputs. If a plugin operation consumes public or private input, its plugin binding must contain the count of the number of consumed public or private inputs per field. Here is an example use of an `assert_equal` plugin that checks that the five inputs are equal to the five next private inputs.

```
/* ... */
@plugin assert_equal;
/* ... */
@begin
  /* ... */
  // declare the function signature with a plugin body
  @function(equal_to_private, @in: 0:5)
    @plugin(assert_equal, private, 0, 5, @private: 0:5);
  /* ... */
  // call the equal_to_private plugin
  @call(equal_to_private, $4 ... $8);
  /* ... */
@end
```

## 4.3 Plugin Types

For some plugins, it is useful to declare additional types that are distinct from ordinary field types. Plugins can define new types by using the `@plugin` directive in the circuit header, which again takes a list of generic parameters. Types declared by plugins have no built-in gates and must be manipulated via its plugin functions.

```
/* ... */
@plugin ring;
// Wire type 0 is the field mod 127
@type field 127;
// Wire type  is the field mod 2**7
@type @plugin(ring, base, 2, exponent, 7);
/* ... */
@begin
  /* ... */
  // interaction with plugin fields is allowed via plugin functions
  @function(int7_mul, @out: 1:1, @in: 1:1, 1:1)
      @plugin(ring, mul, 1);
  @function(int7_add, @out: 1:1, @in: 1:1, 1:1)
      @plugin(ring, add, 1);
  /* ... */
```

```
@end
```

### 4.3.1 RAM Example

Here is an example demonstrating how to use plugins for RAM operations.

```
version 2.0.0;
circuit;
@plugin ram;
@plugin assert_equal;
// Wire type 0 is the field mod 127
@type field 127;
// Wire type 1 is the ram.state plugin type, with addresses and
// values both drawn from field 0.
@type @plugin(ram, state, 0, 0);

@begin
  // Declare RAM operations as abstract functions.
  // ram.init takes a size and returns a RAM state.
  @function(ram_init, @out: 1:1, @in: 0:1)
    @plugin(ram, init, 1);
  // ram.read takes an address and a RAM
  // and returns the value at that address.
  @function(ram_read, @out: 0:1, @in: 0:1, 1:1)
    @plugin(ram, read, 1);
  // ram_write takes a address, value, and RAM
  // writes the value to the address,
  // and returns the updated RAM.
  @function(ram_write, @out: 1:1, @in: 0:1, 0:1, 1:1)
    @plugin(ram, write, 1);

  @function(assert_eq, @in: 0:1, 0:1)
    @plugin(assert_equal, wire, 0);

  $0 <- /* ... */; // address
  $1 <- /* ... */; // value
  $2 <- 0:<5>;
  // Create a new ram.state<field 0, field 0> object.
  $10 <- @call(ram_init, $2);
  // Store $1 at address $0.
  // This produces an updated ram.state object.
  $11 <- @call(ram_write, $0, $1, $10);
  // Load from address $0.
  $3 <- @call(ram_read, $0, $11);
@end
```

# 5 Input Streams

Public and private inputs are provided as separate resources. We have one input file per type and per visibility level (`public_input` or `private_input`). These input files start with the same headers as described in section 2: the version, the resource type (`public_input` or `private_input`), and one type. Then, a sequence of numeric literals representing type elements is provided between `@begin` and `@end` tags. These sequences act as a stream, and certain directives in the circuit consume a value from one of these streams. If values in either stream are exhausted, this is a failure of evaluation validity. If values remain in a stream after processing, then this is also an evaluation invalidity. Here is an example for public and private inputs.

```
version 2.0.0;
public_input;
@type field 7;
@begin
  < 5 >;
@end
```

```
version 2.0.0;
public_input;
@type field 19;
@begin
  < 2 >;
  < 15 >;
@end
```

```
version 2.0.0;
private_input;
@type field 7;
@begin
  < 3 >;
  < 4 >;
@end
```

# Appendix A Binary Syntax

The binary serialization of Circuit-IR will be described here using the open-source FlatBuffers cross-platform serialization library, originally developed by Google. FlatBuffers is a metaformat that specifies the superficial aspects of the syntax, such as representations of literals, structured data and arrays. It moreover supports formal schemas that concretely define what elements (e.g., structures and arrays) can appear in the specific format. FlatBuffers was chosen for the following reasons:

- It offers an existing compact encoding of the format with efficient (de)serialization.
- It is supported by a wide-range of community-based tools and libraries for the most common languages (and is also easy to parse from scratch).

We will use the below FlatBuffers schema to represent `circuit`, `public_input`, and `private_input` ressources. This schema is isomorphic to the Circuit-IR representation presented in Section 3.

```
// This is a FlatBuffers schema.
// See https://google.github.io/flatbuffers/
namespace sieve_ir;

// REGEX used:
// - VERSION_REGEX = "^\d+.\d+.\d+$"
// - STRING_REGEX = "^[a-zA-Z_][\w]*(?:(?:\.|:{2})[a-zA-Z_][\w]*)*$"
// - INTEGER_REGEX = "^[0-9]+$"

// ==== Message types that can be exchanged. ====
union Message {
  Relation,
  PublicInputs,
  PrivateInputs,
}

// The 'version' field must match VERSION_REGEX
// Each string in the 'plugins' list must match STRING_REGEX
table Relation {
  version      :string;
  plugins      :[string];
  types        :[Type];
  conversions  :[Conversion];
  directives   :[Directive];
}

// The 'version' field must match VERSION_REGEX
table PublicInputs {
  version  :string;
  type     :Type;
```

```
  inputs    :[Value];
}

// The 'version' field must match VERSION_REGEX
table PrivateInputs {
  version   :string;
  type      :Type;
  inputs    :[Value];
}

// ==== Helper types ====
// Type element is encoded in a vector of bytes in little-endian
// order. There is no minimum or maximum length; trailing zeros
// may be omitted.
table Value {
  value   :[ubyte];
}

struct Count {
  type_id   :ubyte;
  count     :uint64;
}

// ==== Directive ====
union DirectiveSet {
  Gate,
  Function,
}

table Directive {
  directive   :DirectiveSet;
}

// ==== Conversion ====
struct Conversion {
  output_count   :Count;
  input_count    :Count;
}

// ==== Type ====
union TypeU {
  Field,
  PluginType
}
```

```
table Type {
  element  :TypeU;
}

table Field {
  modulo  :Value;
}

// 'name' and 'operation' must match STRING_REGEX
// Strings of the 'params' list must match either
// STRING_REGEX or INTEGER_REGEX
table PluginType {
  name       :string;
  operation  :string;
  params     :[string];
}

// ==== Gate types ====
table GateConstant {
  type_id    :ubyte;
  out_id     :uint64;
  constant   :[ubyte];
}

table GateAssertZero {
  type_id   :ubyte;
  in_id     :uint64;
}

table GateCopy {
  type_id   :ubyte;
  out_id    :uint64;
  in_id     :uint64;
}

table GateAdd {
  type_id   :ubyte;
  out_id    :uint64;
  left_id   :uint64;
  right_id  :uint64;
}

table GateMul {
  type_id   :ubyte;
  out_id    :uint64;
```

```
  left_id   :uint64;
  right_id  :uint64;
}

table GateAddConstant {
  type_id    :ubyte;
  out_id     :uint64;
  in_id      :uint64;
  constant   :[ubyte];
}

table GateMulConstant {
  type_id    :ubyte;
  out_id     :uint64;
  in_id      :uint64;
  constant   :[ubyte];
}

table GatePublic {
  type_id   :ubyte;
  out_id    :uint64;
}

table GatePrivate {
  type_id   :ubyte;
  out_id    :uint64;
}

// To allocate in a contiguous space all wires between
// first_id and last_id inclusive.
table GateNew {
  type_id    :ubyte;
  first_id   :uint64;
  last_id    :uint64;
}

table GateDelete {
  type_id    :ubyte;
  first_id   :uint64;
  last_id    :uint64;
}

table GateConvert {
  out_type_id    :ubyte;
  out_first_id   :uint64;
```

```
  out_last_id    :uint64;
  in_type_id     :ubyte;
  in_first_id    :uint64;
  in_last_id     :uint64;
}

// ===== Function declaration =====
union FunctionBody {
  Gates,
  PluginBody,
}

table Gates {
  gates  :[Gate];
}

// 'name' and 'operation' must match STRING_REGEX
// Strings of the 'params' list must match either
// STRING_REGEX or INTEGER_REGEX
table PluginBody {
  name           :string;
  operation      :string;
  params         :[string];
  public_count   :[Count]; // Each type_id must be unique
  private_count  :[Count]; // Each type_id must be unique
}

// The 'name' must match STRING_REGEX
table Function {
  // Declare a Function gate as a custom computation or from a plugin
  name          :string;
  output_count  :[Count];
  input_count   :[Count];
  body          :FunctionBody;
}

struct WireRange {
  first_id   :uint64;
  last_id    :uint64;
}

// Invokes a previously defined Function gate
// The 'name' must match STRING_REGEX
table GateCall {
  name    :string;
```

```
  out_ids :[WireRange];
  in_ids  :[WireRange];
}

union GateSet {
  GateConstant,
  GateAssertZero,
  GateCopy,
  GateAdd,
  GateMul,
  GateAddConstant,
  GateMulConstant,
  GatePublic,
  GatePrivate,
  GateNew,
  GateDelete,
  GateConvert,
  GateCall,
}

table Gate {
  gate  :GateSet;
}

// ==== Flatbuffers details ====
// All message types are encapsulated in the FlatBuffers root table.
table Root {
  message  :Message;
}
root_type Root;

// When storing messages to files, this extension and identifier
// should be used.
file_extension "sieve";
file_identifier "siev"; // a.k.a. magic bytes.

// Message framing:
//
// All messages must be prefixed by its size in bytes,
// as a 4-bytes little-endian unsigned integer.
```

As a structured format, the FlatBuffers schema provides a concrete, readable and typed syntax, ensuring syntactic validity. However, it does not provide resource or evaluation validity as it is not a language. Refer to Section 3.8 to a description of syntactic, resource and evaluation validity.

A limitation of the Flatbuffer technology is its 32-bit internal pointer representation, which prevents it from storing buffers larger than approximately 2GB. The IR specifies the following workaround for this limitation.

- The `Root` message may be repeated within a file or stream as manytimes as is necessary: each message holding a portion of the IR resource.
- Each message must be prefixed by its length in bytes, as a 4-byte unsigned little-endian number. (See FinishSizePrefix).
- Each message's `version` attribute must be the same as the first message's `version`. All other attributes must be empty except for the resource's body.

Unfortunately, there is no way for a Flatbuffer to hold a single function which is larger than 2GB.